**PlayWise Music Engine - Detailed Technical Design Document**

**1. High-Level Architecture and Diagrams**

The system is founded upon a modular architectural design, wherein the main.py file serves as the central orchestrator, managing several specialized components. Data integrity is maintained through a central control loop within the main function, which ensures all data structures remain synchronized during user interactions.

**System Components:**

* **Playlist**: The foundational data structure for managing song lists, implemented as a Doubly Linked List.
* **PlaybackHistory**: A LIFO (Last-In, First-Out) stack, designed to facilitate the reversal of recently played songs.
* **RatingBST**: A Binary Search Tree, utilized for indexing songs based on a 1-5 star rating system.
* **SongLookup**: A hash map providing O(1) song lookups by ID and title.
* **PlaylistSwitcher**: A component that manages state and playback position across multiple playlists by employing a HashMap of stacks.
* **VolumeNormalizer**: A module dedicated to a specific aggregation and adjustment task, focused on audio levels.
* **SnapshotDashboard**: A reporting module that consolidates data from multiple sources to provide a comprehensive system overview.

**UML Diagram:** A screenshot of a computer screen

AI-generated content may be incorrect.

**2. Data Structure and Algorithm Justification**

**Data Structure Selection Trade-offs**

* **Doubly Linked List for Playlist (playlist\_engine.py)**:
  + **Trade-off**: While indexed access (e.g., delete\_song(index)) has a time complexity of O(n), operations such as appending to the end (add\_song) or reversing the list are highly efficient. This structure is particularly well-suited for real-world playlist modifications, such as moving a song, which can be accomplished without extensive memory shifting.
  + **Justification**: The primary use cases, including song addition, playback navigation, reordering, and reversal, are effectively supported with low time complexity. The O(n) cost for deletion by index is deemed acceptable, given that typical user playlists are of a manageable size.
* **Stack for Playback History (playback\_history.py)**:
  + **Trade-off**: The stack, by its nature as a LIFO structure, permits the undoing of only the most recent action. It does not provide a mechanism to reverse a specific action that occurred earlier in the sequence without also undoing all subsequent actions.
  + **Justification**: This is the optimal data structure for the "undo last play" feature. The O(1) time complexity of push and pop operations ensures that the history functionality is both exceptionally fast and memory efficient.
* **Binary Search Tree for Ratings (SongRating\_tree.py)**:
  + **Trade-off**: The performance of a BST is contingent upon its balance. Given the limited range of 5 unique ratings, the tree remains shallow, meaning the worst-case time complexity of O(k) (where k represents the number of unique ratings) is nearly as efficient as the average-case O(logk).
  + **Justification**: The BST offers a clear, hierarchical method for organizing and retrieving songs based on ratings. Insertion, deletion, and searching operations are highly efficient. Furthermore, the structure is easily traversable, which is beneficial for tasks such as the rating count function within the snapshot module.
* **Hash Map for Song Lookup (instant\_song\_lookup.py)**:
  + **Trade-off**: While the theoretical worst-case performance can degrade to O(n) due to hash collisions, this is a rare occurrence in practice with a well-designed hashing function.
  + **Justification**: For features requiring instantaneous lookups by a song's ID or title, a hash map is the superior choice. Its average-case time complexity for both insertion and retrieval is O(1), which is essential for a fast and responsive user experience.

**Algorithm Choice Justification**

* **Merge Sort (mergesort.py)**:
  + **Justification**: Merge sort was selected for the playlist sorting functionality. Its time complexity is a consistent O(nlogn) across all cases (best, average, and worst), guaranteeing predictable performance. The algorithm is also stable, which ensures that songs with identical sorting keys (e.g., two songs of the same duration) will maintain their original relative order. This characteristic is vital for preserving playlist integrity and meeting user expectations. The algorithm's space complexity is O(n), attributed to the temporary arrays created during the merging process, representing a standard and acceptable trade-off for its performance and stability guarantees.

**3. Pseudocode for Major Algorithms**

***i. Merge Sort (from `mergesort.py`)***

function mergeSort(array):

if length of array <= 1:

return array

mid = length of array // 2

left = mergeSort(array[0:mid])

right = mergeSort(array[mid:])

return merge(left, right)

function merge(left, right):

result = empty array

while left and right are not empty:

if left[0] <= right[0]:

append left[0] to result

remove left[0] from left

else:

append right[0] to result

remove right[0] from right

append any remaining elements from left to result

append any remaining elements from right to result

return result

***ii. Playback History (from `playback\_history.py`)***

class PlaybackHistory:

initialize empty list history

function add\_song(song):

append song to history

function get\_recent(n):

return last n songs from history

function clear\_history():

clear the history list

***iii. Playlist Engine (from `playlist\_engine.py`)***

class PlaylistEngine:

initialize playlist as empty list

function add\_song(song):

append song to playlist

function remove\_song(song):

remove song from playlist if it exists

function get\_next\_song():

return the next song in the playlist

function shuffle\_playlist():

randomly shuffle the playlist

function sort\_playlist(criteria):

sort playlist based on criteria (e.g., by title, artist, rating)

***iv. Song Rating Tree (from `SongRating\_tree.py`)***

class SongNode:

initialize song, rating, left, right

class SongRatingTree:

root = None

function insert(song, rating):

if root is None:

root = new SongNode(song, rating)

else:

recursively insert into left or right subtree based on rating

function find(song):

search for song in the tree

function get\_top\_rated(n):

perform in-order traversal to get top n rated songs

***v. Instant Song Lookup (from `instant\_song\_lookup.py`)***

class SongLookup:

initialize song dictionary

function add\_song(song):

add song to dictionary with relevant details

function lookup(song\_name):

return song details if song\_name exists in dictionary

**4. Time and Space Complexity Annotations**

The provided Python files include in-line annotations detailing the time and space complexity for most core functions, including:

* **Playlist.add\_song**: Time: O(1), Space: O(1)**Error! Filename not specified.**
* **Playlist.reverse\_playlist**: Time: O(n), Space: O(1)**Error! Filename not specified.**
* **Playlist.sort\_playlist (using merge\_sort)**: Time: O(nlogn), Space: O(n)**Error! Filename not specified.**
* **PlaybackHistory.push**: Time: O(1), Space: O(1)**Error! Filename not specified.**
* **RatingBST.insert\_song**: Average Time: O(logk), Worst: O(k), where k is the number of unique ratings (5).
* **SongLookup.get\_by\_id**: Average Time: O(1), Worst: O(n)**Error! Filename not specified.**

**5. Test Case Results**

Each of the provided Python files contains a test() or main() function that demonstrates its functionality. The expected outcomes are as follows:

* **playlist\_engine.py (test1)**: The test verifies the successful creation of a playlist, the deletion and movement of a song, and the correct reversal of the playlist. The print\_playlist() output visually confirms these operations.
* **instant\_song\_lookup.py (test1)**: This test confirms that songs are added and synchronized correctly. Lookups by song ID and title are successful, returning the correct songs. The sorting functionality is also verified, demonstrating correct sorting by title and duration.
* **playback\_history.py (main)**: The test simulates song playback, populating the playback stack. The undo\_last\_play function is then shown to successfully re-add the last played song to the beginning of the playlist.
* **volumecontrol.py**: A playlist with varied volumes is created. The normalize() function is executed, and the print\_volumes() output confirms that all songs now have an adjusted\_volume equal to the calculated average.
* **SongRating\_tree.py (main)**: This test involves inserting songs with ratings into the BST. The program successfully searches for and prints songs with a specific rating. A song is then deleted from the BST, and the print\_all\_ratings function confirms its removal.
* **main.py**: This file provides an interactive console application that validates the seamless integration of all modules. Every menu option, from adding a new song to viewing the system snapshot, tests the corresponding functionality and data synchronization across the entire system.